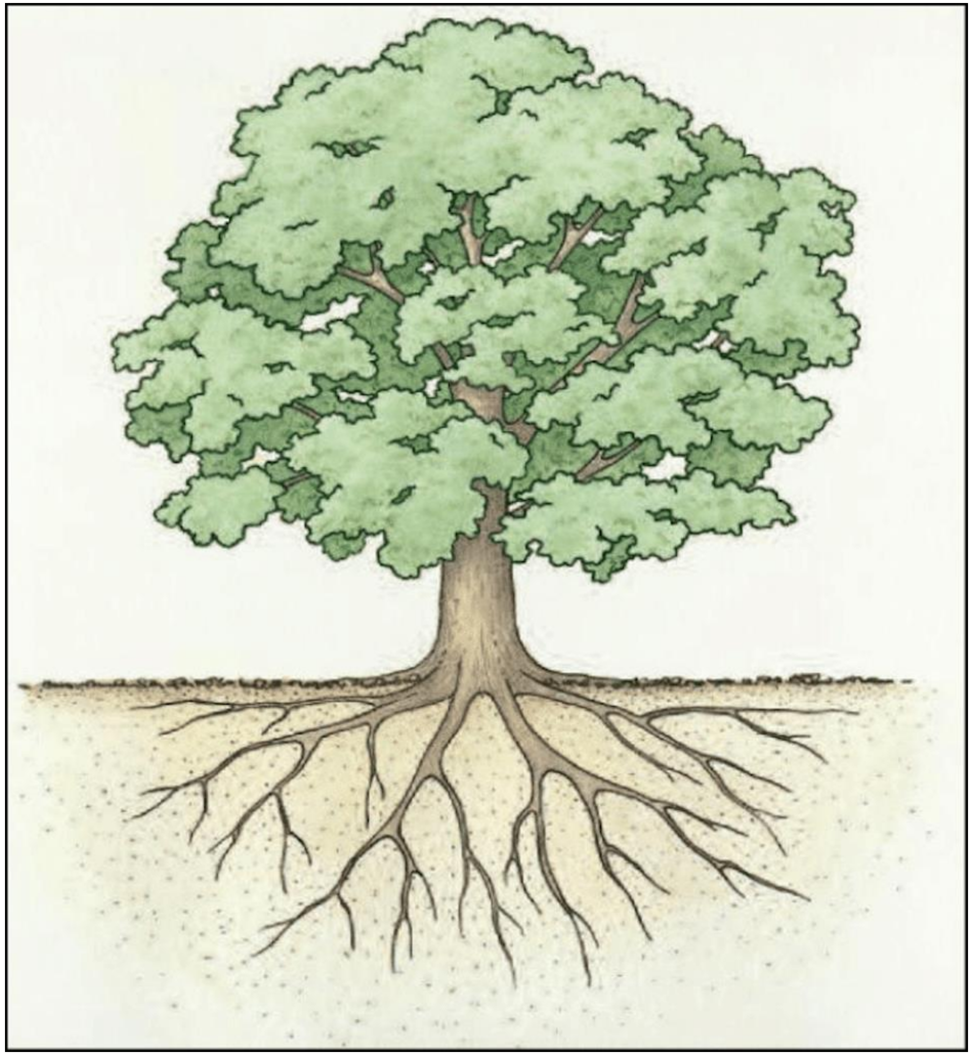
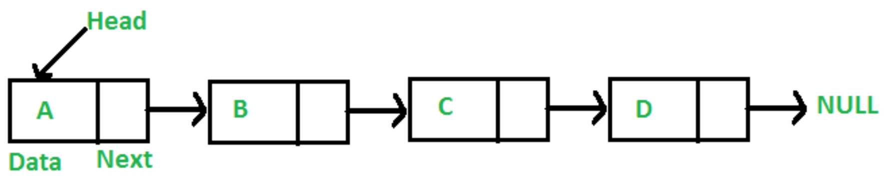
Trees



Trees

Linked list with 4 nodes



- Linked lists have $O(n)$ running time for many operations (too slow!)
  - Find(elem), delete(elem), findMax, findMin, etc.
- Trees are similar to linked lists, but have $O(logn)$ running time for most operations
- A tree is a *collection of nodes* (like a linked list) but with hierarchical structure
- **Recursive definition:** Consists of a *root* r and zero or more non-empty subtrees
- The root of each subtree is a *child* of r, and r is the *parent* of each subtree root
- Each node may have an arbitrary number of children (possibly zero). Nodes with 0 children are *leaves*
- A *path* from node n_1 to n_k is a sequence of nodes $n_1, n_2, \ldots n_k$ such that n_i is the parent of n_{i+1} for $1 \leq i < k$. *Length* of path is number of edges in path.
- For any node n_i, the *depth* of n_i is length of unique path from root to n_i
- Root is at depth 0.
- The *height* of n_i is the length of the longest path from n_i to a leaf

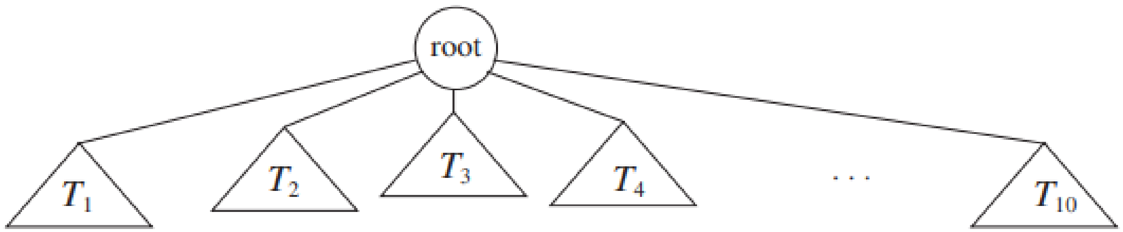


Figure 4.1 Generic tree

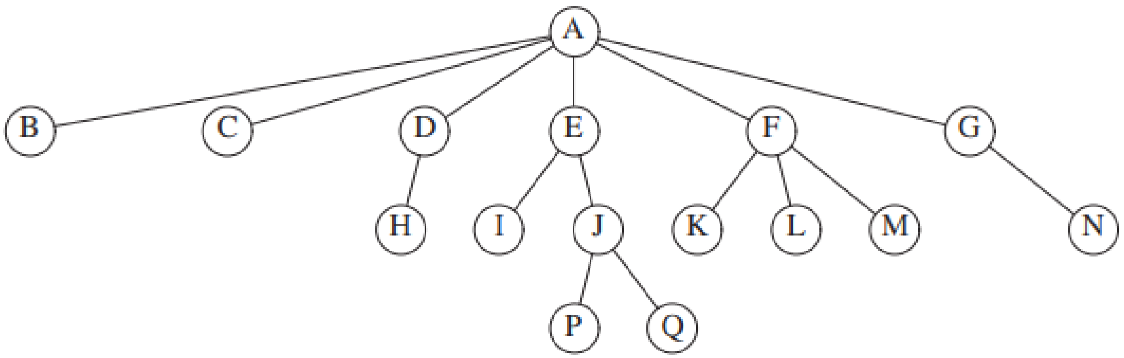


Figure 4.2 A tree

Binary trees

- A **binary tree** is a tree in which no node can have more than two children
- Figure 4.11: a binary tree and consisting of a root and two subtrees T_L and T_R, both of which are possibly empty
- Figure 4.13: node definition for a Binary tree node. Similar to a doubly-linked list, in that it contains two pointers to other nodes and a data element

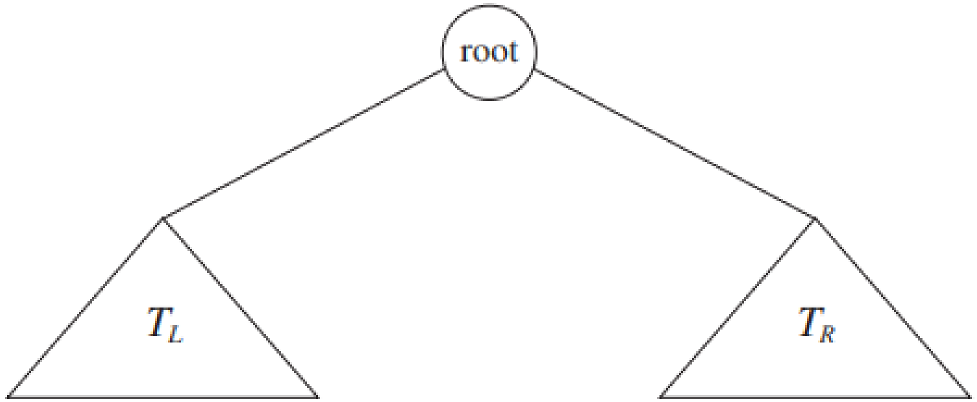


Figure 4.11 Generic binary tree

```
struct BinaryNode
{
    Object      element;      // The data in the node
    BinaryNode *left;         // Left child
    BinaryNode *right;        // Right child
};
```

Figure 4.13 Binary tree node class (pseudocode)

Application of Binary Tree: expression tree

- Figure 4.14: an example of an expression tree
- *Leaves* of the expression tree are operands (numbers or constants) and the other nodes contain operators (+, *, etc.)
- Can evaluate expression tree by recursively evaluating left and right subtrees. Here:
  - Left subtree: a+b*c
  - Right subtree: (d*e+f)*g

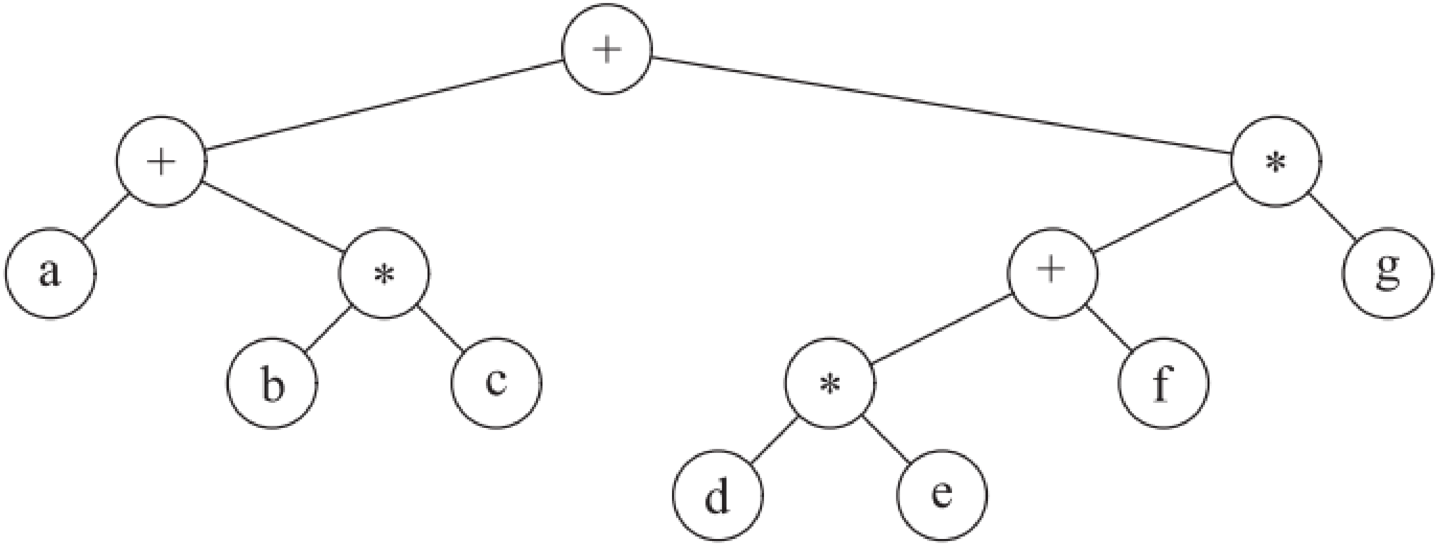
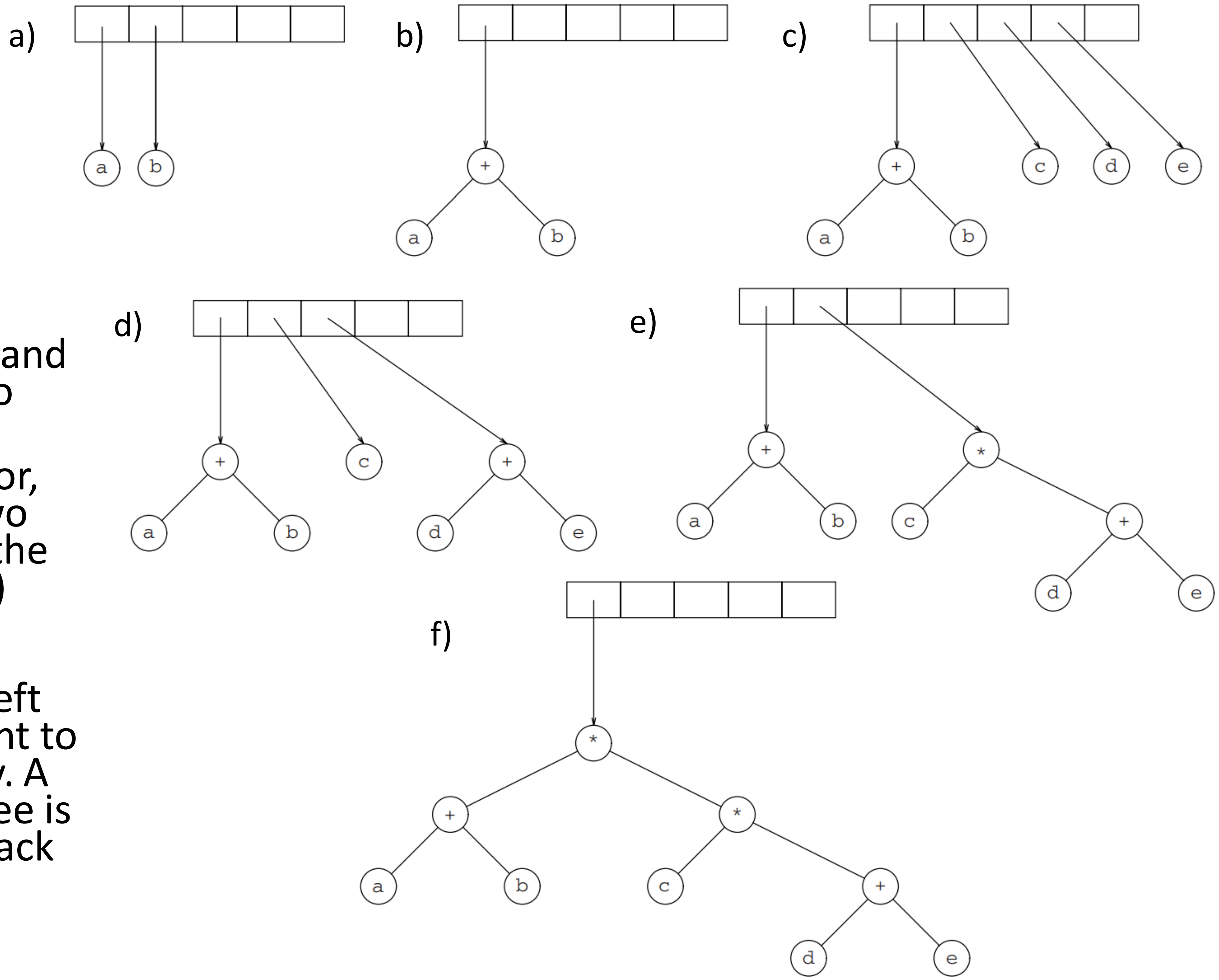


Figure 4.14 Expression tree for `(a + b * c) + ((d * e + f ) * g)`

Constructing expression tree from postfix

- Postfix expression: `ab+cde+**`
- Algorithm:
- Read expression one symbol at a time from left to right
  - If symbol is operand, create one-node tree and push pointer to it unto the stack
  - If symbol is an operator, we pop pointers to two trees T_1 and T_2 from the stack (T_1 popped first) and form a new tree whose root is the operator and whose left and right children point to T_1 and T_2 respectively. A pointer to this new tree is then pushed to the stack



Binary Search Tree (BST)

- An important application of binary trees is their use in searching
- Assume each node in the tree stores an item (integer)
- **BST property:** for every node X in tree, values of all items in $X's$ left subtree are smaller than $X's$ value, and values of all items in $X's$ right subtree are larger than $X's$ value (will deal with duplicates later)

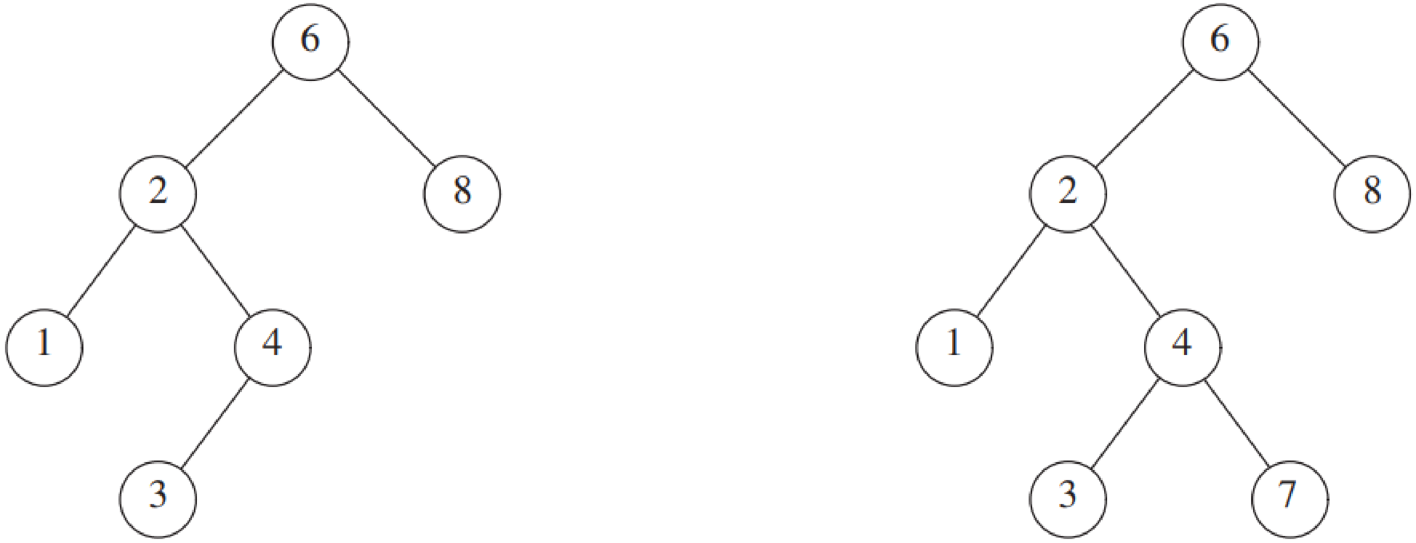


Figure 4.15 Two binary trees (only the left tree is a search tree)

BST code skeleton

```
template <typename Comparable>
class BinarySearchTree
{
  public:
    BinarySearchTree( );
    BinarySearchTree( const BinarySearchTree & rhs );
    BinarySearchTree( BinarySearchTree && rhs );
    ~BinarySearchTree( );

    const Comparable & findMin( ) const;
    const Comparable & findMax( ) const;
    bool contains( const Comparable & x ) const;
    bool isEmpty( ) const;
    void printTree( ostream & out = cout ) const;

    void makeEmpty( );
    void insert( const Comparable & x );
    void insert( Comparable && x );
    void remove( const Comparable & x );

    BinarySearchTree & operator=( const BinarySearchTree & rhs );
    BinarySearchTree & operator=( BinarySearchTree && rhs );

  private:
    struct BinaryNode
    {
        Comparable element;
        BinaryNode *left;
        BinaryNode *right;

        BinaryNode( const Comparable & theElement, BinaryNode *lt, BinaryNode *rt )
          : element{ theElement }, left{ lt }, right{ rt } { }

        BinaryNode( Comparable && theElement, BinaryNode *lt, BinaryNode *rt )
          : element{ std::move( theElement ) }, left{ lt }, right{ rt } { }
    };

    BinaryNode *root;

    void insert( const Comparable & x, BinaryNode * & t );
    void insert( Comparable && x, BinaryNode * & t );
    void remove( const Comparable & x, BinaryNode * & t );
    BinaryNode * findMin( BinaryNode *t ) const;
    BinaryNode * findMax( BinaryNode *t ) const;
    bool contains( const Comparable & x, BinaryNode *t ) const;
    void makeEmpty( BinaryNode * & t );
    void printTree( BinaryNode *t, ostream & out ) const;
    BinaryNode * clone( BinaryNode *t ) const;
};
```

BST method: `contains`

- `contains` returns true if there is a node in tree T that has item X, or false otherwise
- Line 8: if tree is empty, return `false`
- Line 10: if the element we are searching for is less than current node's element, call recursively on left child
- Line 12: if element we are searching for is greater than current node's element, call recursively on right child
- Line 15: otherwise, return `true`
- Order of these if/else statements is important. Must test for empty tree first, otherwise we will be accessing data on a nullptr (generates runtime exception)

```
1  /**
2   * Internal method to test if an item is in a subtree.
3   * x is item to search for.
4   * t is the node that roots the subtree.
5   */
6  bool contains( const Comparable & x, BinaryNode *t ) const
7  {
8      if( t == nullptr )
9          return false;
10     else if( x < t->element )
11         return contains( x, t->left );
12     else if( t->element < x )
13         return contains( x, t->right );
14     else
15         return true;    // Match
16 }
```

Figure 4.18 contains operation for binary search trees

BST method: `findMin`/`findMax`

- `findMin` and `findMax` return pointers to node containing smallest and largest elements in tree, respectively.
- Can be implemented recursively or non-recursively
- `findMin`: follow left children until we reach a nullptr
- `findMax`: follow right children until we reach a `nullptr`

```
/**
 * Internal method to find the smallest item in a subtree t.
 * Return node containing the smallest item.
 */
BinaryNode * findMin( BinaryNode *t ) const
{
    if( t == nullptr )
        return nullptr;
    if( t->left == nullptr )
        return t;
    return findMin( t->left );
}
```

Figure 4.20 Recursive implementation of `findMin` for binary search trees

```
/**
 * Internal method to find the largest item in a subtree t.
 * Return node containing the largest item.
 */
BinaryNode * findMax( BinaryNode *t ) const
{
    if( t != nullptr )
        while( t->right != nullptr )
            t = t->right;
    return t;
}
```

Figure 4.21 Nonrecursive implementation of `findMax` for binary search trees

BST method: `insert`

- To insert element X into tree T, proceed down tree as with `contains`
  - If X is already in tree, do nothing
  - Otherwise, insert X at the last spot on the path traversed
- Example: inserting 5
  - Traverse tree searching for element 5 until we reach a `nullptr`, and then insert 5 at that position

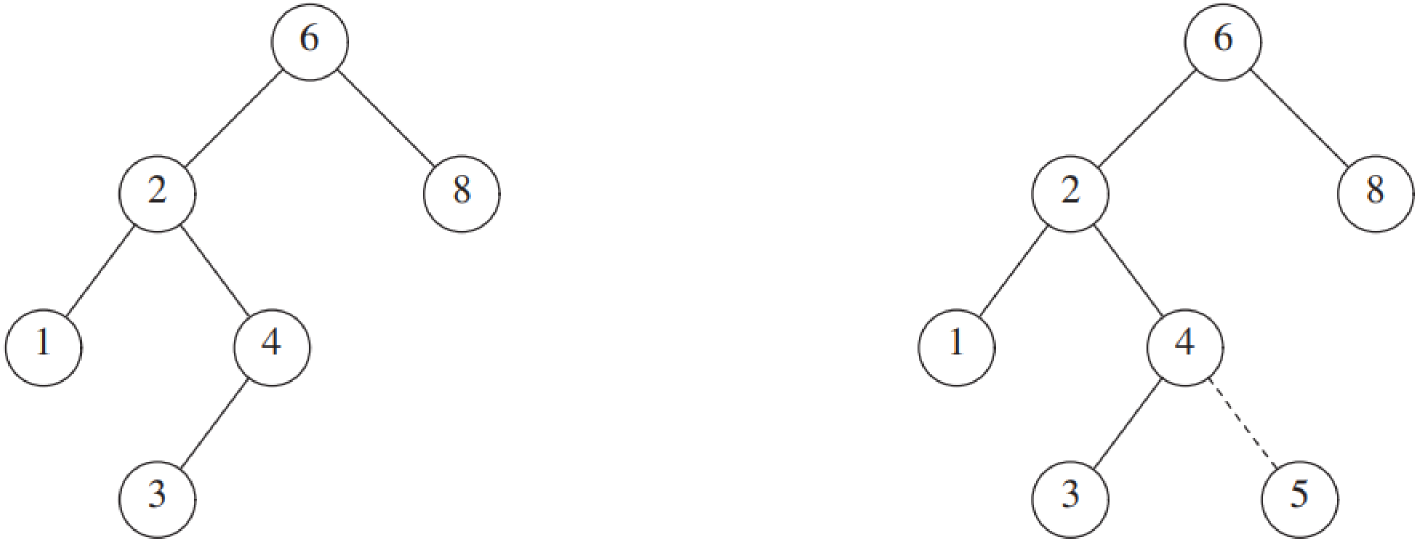


Figure 4.22 Binary search trees before and after inserting 5

BST method: `insert`

a) public insert method

```cpp
/**
 * Insert x into the tree; duplicates are ignored.
 */
void insert( const Comparable & x )
{
    insert( x, root );
}

```

b) private insert method (lvalue)

```cpp
/**
 * Internal method to insert into a subtree.
 * x is the item to insert.
 * t is the node that roots the subtree.
 * Set the new root of the subtree.
 */
void insert( const Comparable & x, BinaryNode * & t )
{
    if( t == nullptr )
        t = new BinaryNode{ x, nullptr, nullptr };
    else if( x < t->element )
        insert( x, t->left );
    else if( t->element < x )
        insert( x, t->right );
    else
        ;  // Duplicate; do nothing
}

```

c) private insert method (rvalue)

```cpp
/**
 * Internal method to insert into a subtree.
 * x is the item to insert by moving.
 * t is the node that roots the subtree.
 * Set the new root of the subtree.
 */
void insert( Comparable && x, BinaryNode * & t )
{
    if( t == nullptr )
        t = new BinaryNode{ std::move( x ), nullptr, nullptr };
    else if( x < t->element )
        insert( std::move( x ), t->left );
    else if( t->element < x )
        insert( std::move( x ), t->right );
    else
        ;  // Duplicate; do nothing
}
```

Figure 4.23 Insertion into a binary search tree

BST method: `remove`

- To remove a node, first find it.
- Once node has been found, there are three possibilities:
  - 1) node is a leaf. It can be deleted immediately without affecting rest of tree
  - 2) node has one child. It can be deleted and set deleted node's parent pointer to deleted node's child
  - 3) node has two children. General strategy is to *replace node's data with smallest data of it's right subtree* and then recursively delete that node (which is now empty)

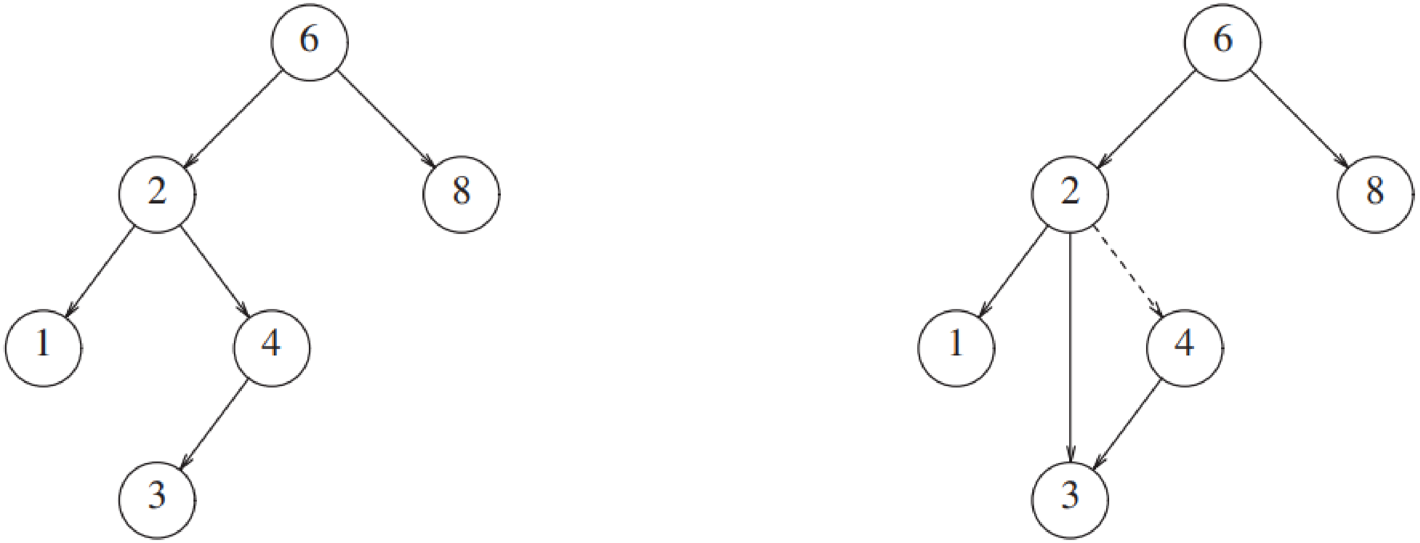


Figure 4.24 Deletion of a node (4) with one child, before and after

BST method: `remove`

```
/**
 * Remove x from the tree. Nothing is done if x is not found.
 */
void remove( const Comparable & x )
{
    remove( x, root );
}
```

```
/**
 * Internal method to remove from a subtree.
 * x is the item to remove.
 * t is the node that roots the subtree.
 * Set the new root of the subtree.
 */
void remove( const Comparable & x, BinaryNode * & t )
{
    if( t == nullptr )
        return;   // Item not found; do nothing
    if( x < t->element )
        remove( x, t->left );
    else if( t->element < x )
        remove( x, t->right );
    else if( t->left != nullptr && t->right != nullptr ) // Two children
    {
        t->element = findMin( t->right )->element;
        remove( t->element, t->right );
    }
    else
    {
        BinaryNode *oldNode = t;
        t = ( t->left != nullptr ) ? t->left : t->right;
        delete oldNode;
    }
}
```

Figure 4.26 Deletion routine for binary search trees

Case 3) node has two children. General strategy is to *replace node's data with smallest data of it's right subtree* and then recursively delete that node (which is now empty)

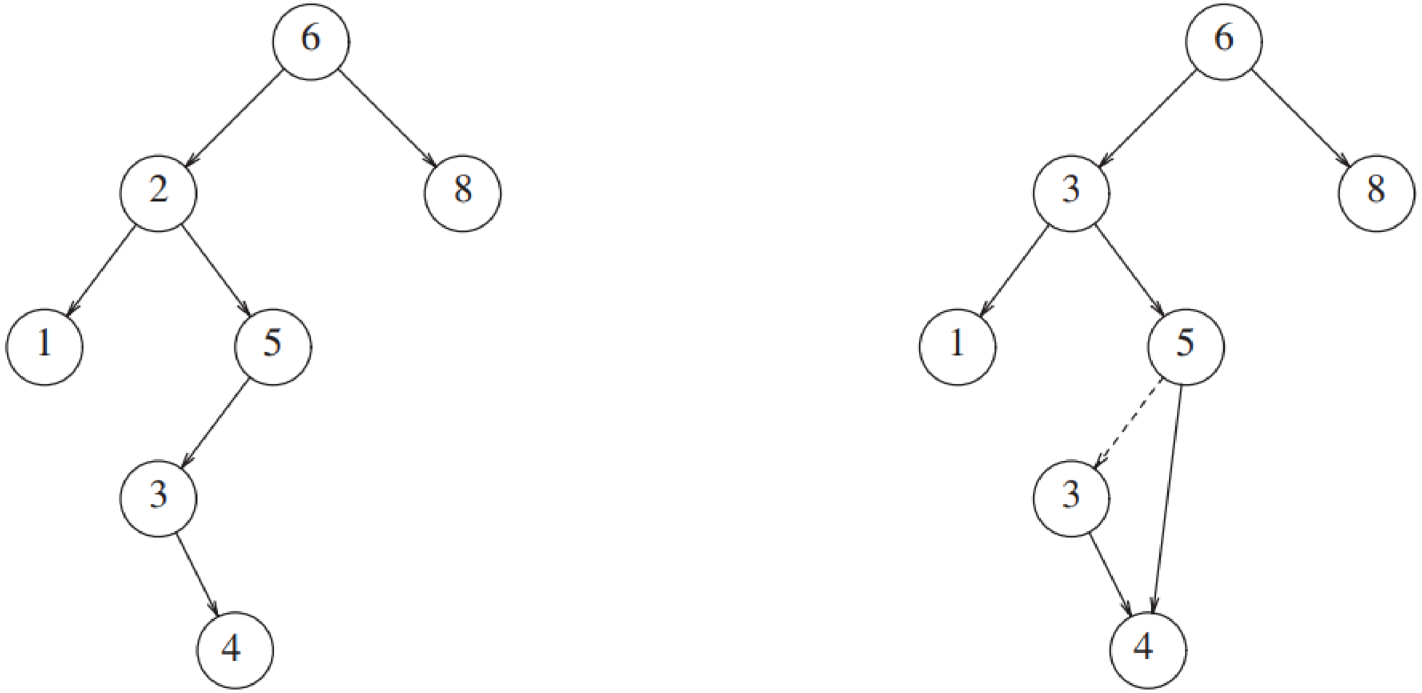
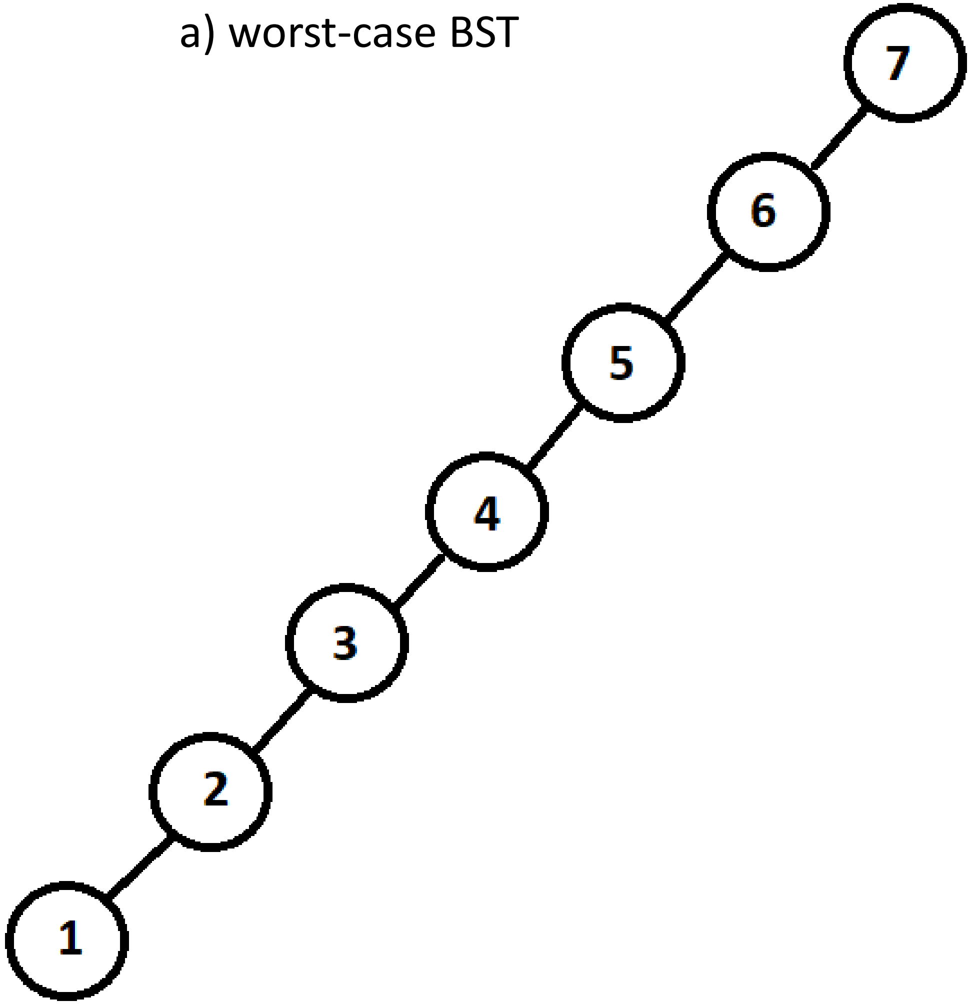


Figure 4.25 Deletion of a node (2) with two children, before and after

BST worst-case

- If sequence from which tree is built is in sorted or reverse-sorted order, BST will devolve into a linked list (worst-case BST)
- Example: inserting [7,6,5,4,3,2,1] yields tree in (a)
- All operations (findMin, findMax, contains, remove, insert) will take $O(n)$ in this worst-case tree

a) worst-case BST



BST average case

- All operations except `makeEmpty` and `clone` take $O(log n)$ in an average-case BST
- We descend one level in constant time, which cuts the number nodes in half
  - Running time of all operations is $O(d)$, where d is the depth of the node containing the accessed item
- To see this, just imagine a 'full' BST with n nodes. Because the number of nodes doubles on each level down, it will have max depth $\log_2(n)$

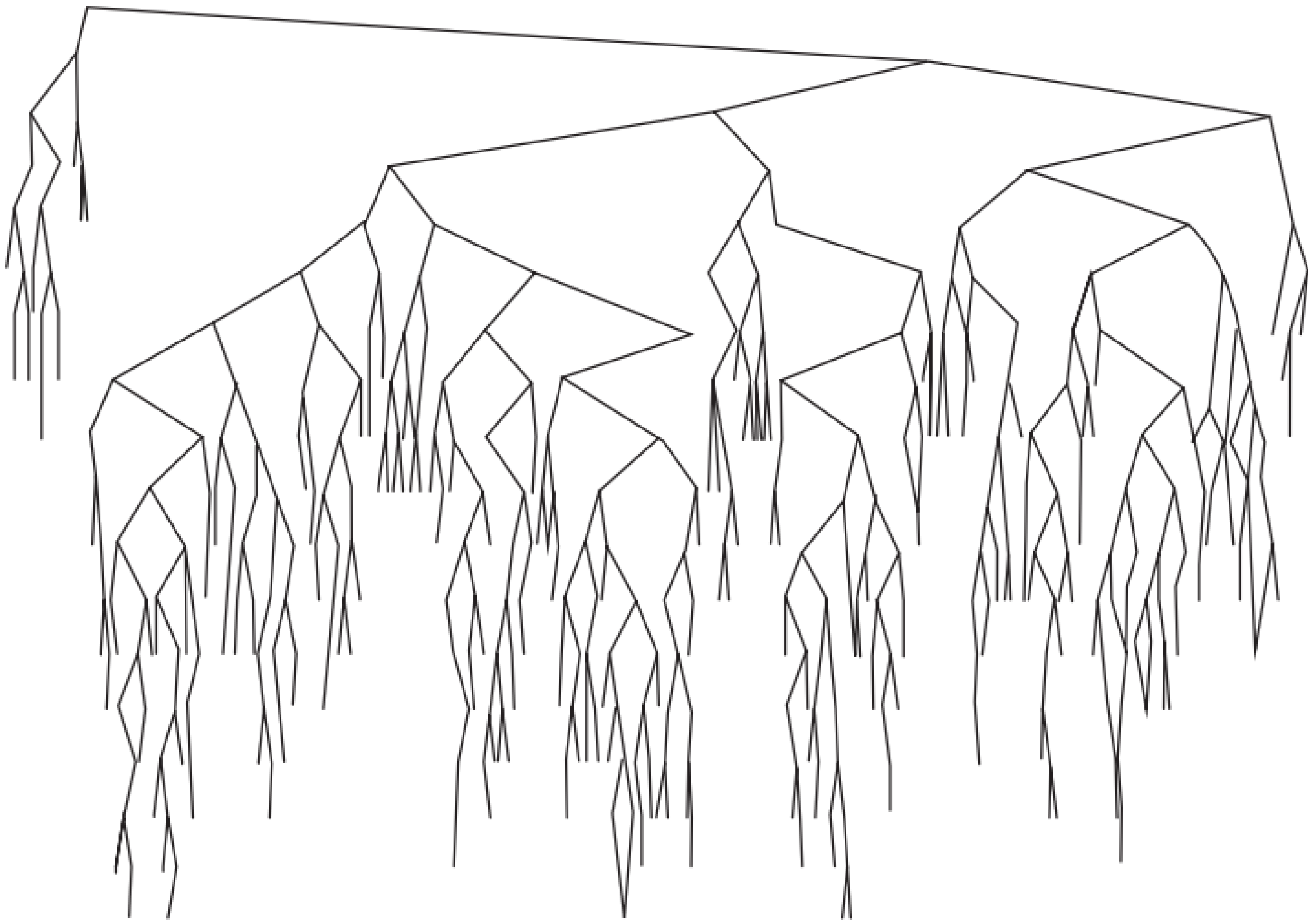
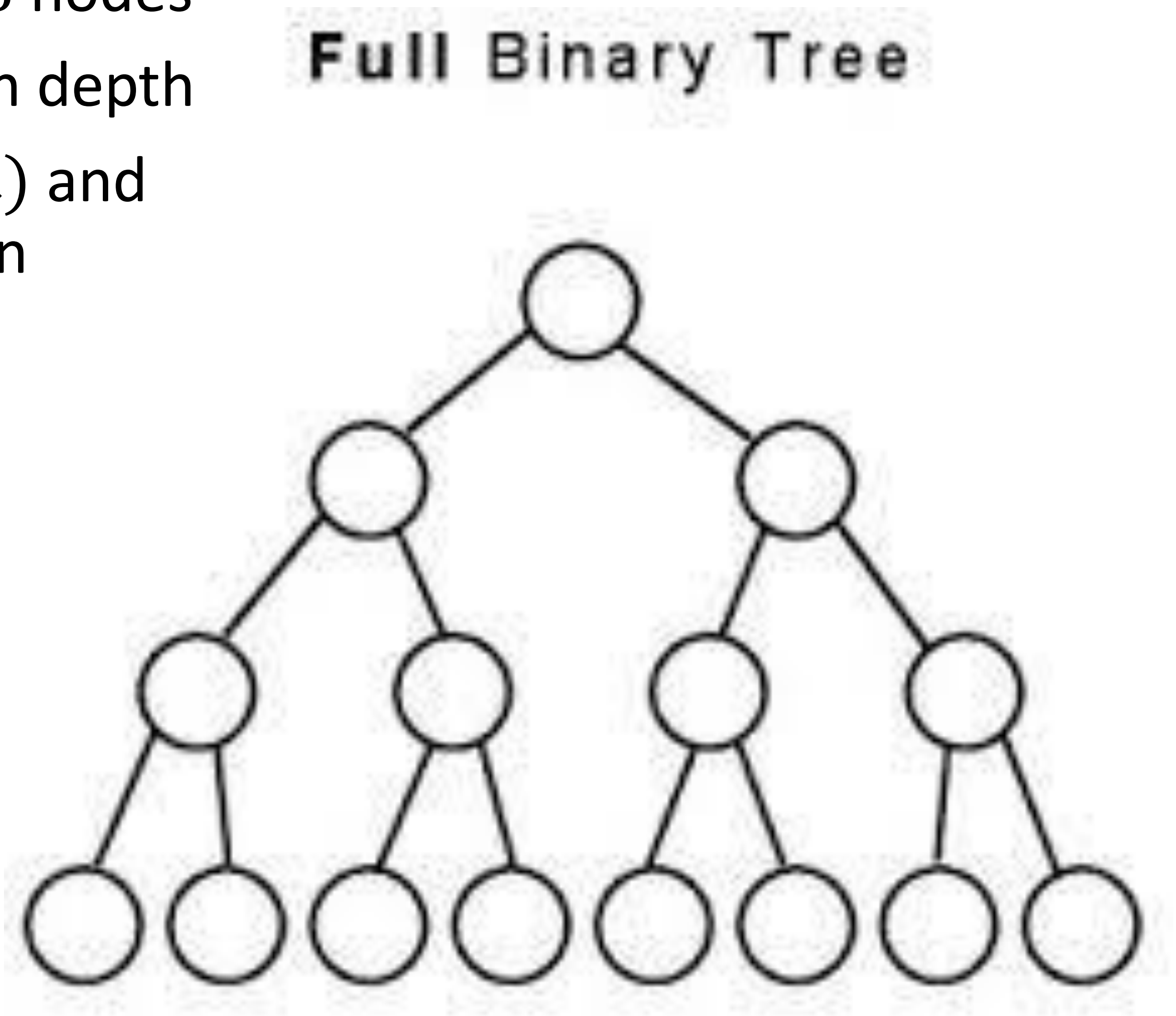


Figure 4.29 A randomly generated binary search tree

- Example: Binary tree with $n = 15$ nodes
- Number of nodes doubles at each depth
- Therefore, max depth is $O(log_2 n)$ and all BST operations are $O(logn)$ (in average case)



BST in C++ STL

- Used to implement the set and map ADTs

Set

Sets are containers that store unique elements following a specific order.

In a `set`, the value of an element also identifies it (the value is itself the ***key***, of type `T`), and each value must be unique. The value of the elements in a `set` cannot be modified once in the container (the elements are always const), but they can be inserted or removed from the container.

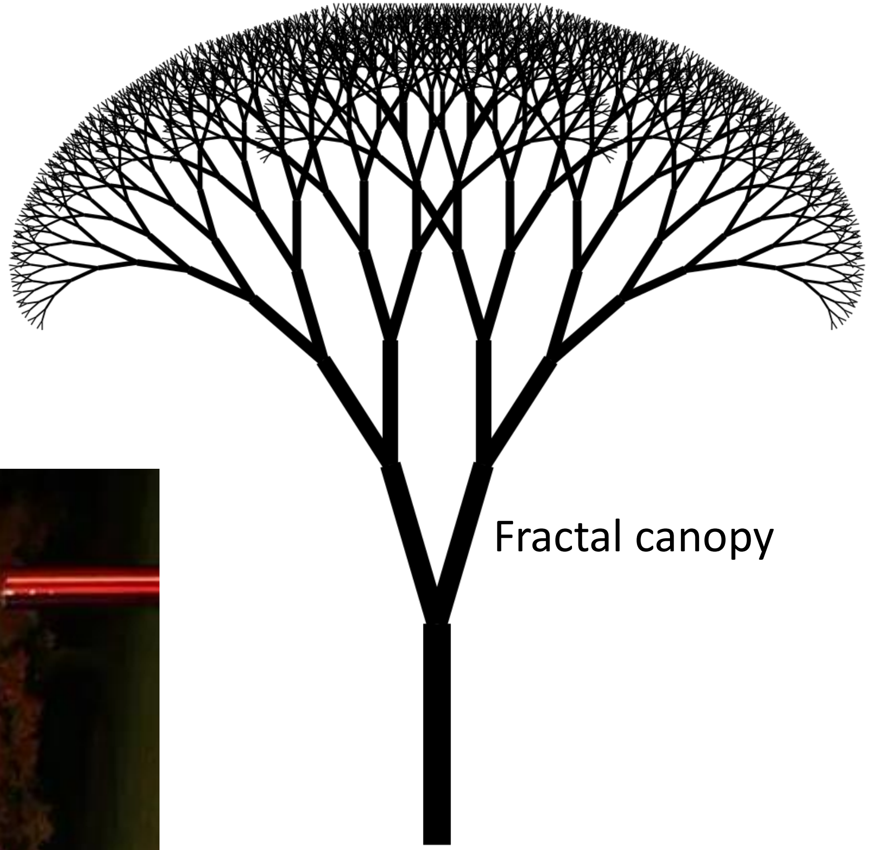
Map

Maps are associative containers that store elements formed by a combination of a ***key value*** and a ***mapped value***, following a specific order.

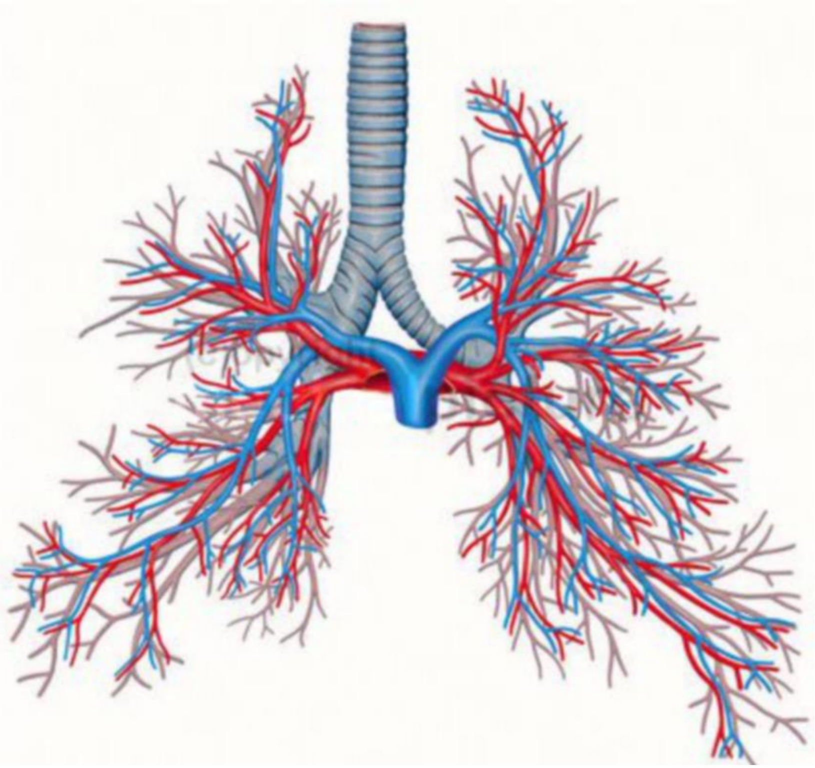
In a `map`, the ***key values*** are generally used to sort and uniquely identify the elements, while the ***mapped values*** store the content associated to this ***key***. The types of ***key*** and ***mapped value*** may differ, and are grouped together in member type `value_type`, which is a pair type combining both:

trees in nature

https://en.wikipedia.org/wiki/Fractal_canopy



Fractal canopy



Pulmonary tree

tree

Electrical breakdown

Viscous fingering (Saffman-Taylor instability)